实验五 进程和线程同步

# 实验目的

加深对进程概念的理解，认识并发执行的实质，分析进程征用资源的现象，学习进程、线程互斥的方法。

# 实验内容

线程是计算机中独立运行的最小单位，运行时占用很少的系统资源。由于每个线程占用的CPU时间是由系统分配的，因此可以把线程看成操作系统分配CPU时间的基本单位。在用户看来，多个线程是同时执行的，但从操作系统调度上看，各个线程是交替执行的。系统不停地在各个线程之间切换，每个线程只有系统分配给它的时间片内才能取得CPU的控制权，执行线程中的代码。

注意，这里只是针对单CPU单核的情况，在多CPU多核的主机上，多个线程是可以同时运行的。Linux操作系统是支持多线程的，他在一个进程内生成了许多个线程。一个进程可以拥有一至多个线程。

虽然线程在进程内部共享地址空间、打开的文件描述符等资源。但是线程也有其私有的数据信息，包括：

线程号：（thread ID）：每个线程都有一个唯一的线程号与其对应

寄存器（包括程序计数器和堆栈指针）、堆栈、信号掩码、优先级、线程私有的存储空间。Linux系统支持POSIX多线程接口，称为pthread(Posix Thread的简称)。编写Linux下的多线程应用程序，需要使用头文件pthread.h，链接时需要使用库libpthread.a。

创建线程：

线程创建函数pthread\_create

前面的程序实例都是单线程的。单线程的程序都是按照一定的顺序执行的，如果在主线程里面创建线程，程序就会在创建线程的地方产生分支，变成两个程序执行。这似乎和多进程一样，其实不然。子进程是通过拷贝父进程的地址空间来实现的；而线程与进程内的线程共享程序代码，一段代码可以同时被多个线程执行。

线程的创建通过函数pthead\_create来完成，该函数的声明如下：

#include<pthread.h>

int pthread\_create(pthread\_t \*thread, pthread\_attr\_t \* attr,void\*(\*start\_routine)(void \*),void

\*arg);

注意：线程创建成功时，pthread\_create函数返回0，若不为0则说明创建进程失败。常见的错误码为EAGAIN和EINVAL。前者表示系统限制创建新的线程，例如，线程数目过多；后者表示第2个参数代表的线程属性非法。线程创建成功

后，新创建的线程开始运行第3个参数所指向的函数，原来的线程继续运行。

pthead\_create函数的第2个参数attr是一个指向pthread\_attr\_t结构体的指针，该结构体指明待创建线程的属性。

创建线程的其他系统函数

|  |  |
| --- | --- |
| 函数 | 说明 |
| pthread\_t pthread\_self（void） | 获取本线程的线程ID |
| int pthread\_equal(pthread\_t thread，pthread\_thread2) | 判断两个线程ID是否指向同一线程 |
| int pthread\_once(pthread\_once\_t \*once\_control,void(\*init\_routine)(void)) | 用来保证init\_routine线程函数在进程中仅  执行一次 |

下面通过5-1讲述线程的创建过程。例5-1 createthread.c

#include <stdio.h> #include <stdlib.h> #include <unistd.h> #include <pthread.h> int \* thread(void \* arg)

{

pthread\_t newthid; newthid = pthread\_self();

printf("this is a new thread, thread ID = %d\n", newthid); return NULL;

}

int main(void)

{

pthread\_t thid;

printf("main thread ,ID is %d\n",pthread\_self()); if(pthread\_create(&thid, NULL, (void \*)thread, NULL) != 0) {

printf("thread creation failed\n"); exit(1);

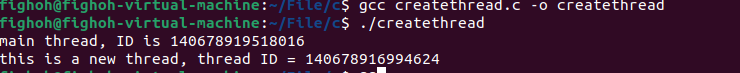
}

exit(0);

}

**编译并运行$gcc-o createthread createthread.c –lpthread**

**$./createthread**

程序首先打印出主线程的ID，然后打印新创建线程的ID。

在某些情况下，函数执行次数要被限制为一次，这种情况下就要使用 pthread\_once函数。下面通过例5-2说明。该实例中创建两个线程，两个线程分别通过pthread\_once调用同一个函数，结果被调用的函数只被执行了一次。

例5-2 oncerun.c #include <stdio.h>

#include <pthread.h>

pthread\_once\_t once = PTHREAD\_ONCE\_INIT; void run(void)

{

printf("Fuction run is running in thread %d\n",pthread\_self());

}

void \* thread1(void \*arg)

{

pthread\_t thid=pthread\_self(); printf("Current thread's ID is %d\n", thid); pthread\_once(&once,run);

printf("thread1 ends\n");

}

void \* thread2(void \*arg)

{

pthread\_t thid=pthread\_self(); printf("Current thread's ID is %d\n", thid); pthread\_once(&once, run); printf("thread2 ends\n");

}

int main(void)

{

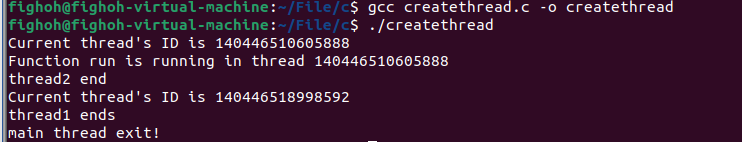
pthread\_t thid1,thid2;

pthread\_create(&thid1,NULL,thread1,NULL); pthread\_create(&thid2,NULL, thread2,NULL); sleep(3);

printf("main thread exit! \n"); exit(0);

}

## 请编译并运行，通过结果分析线程的执行情况。

由于受到pthread\_once的限制run函数只被执行了一次

**线程同步**

线程最大的特点就是资源的共享性，然而资源共享中的同步问题是多线程编程的难点。Linux系统提供了多种方式处理线程间的同步问题，其中最常用的有互斥锁、条件变量和异步信号。下面将重点介绍这3种同步技术的使用。

## 互斥锁

互斥锁通过锁机制来实现线程的同步。在同一时刻它通常只允许一个线程执行关键部分的代码。下表列举了操作互斥锁的几个函数。这些函数均声明在头文件pthread.h中。

互斥锁函数

|  |  |
| --- | --- |
| 函数 | 功能 |
| pthread\_mutex\_int函数 | 初始化一个互斥锁 |
| pthread\_mutex\_destroy函数 | 注销一个互斥锁 |
| pthread\_mutex\_lock函数 | 加锁，如果不成功，阻塞等待 |
| pthread\_mutex\_unlock函数 | 解锁 |
| pthread\_mutex\_trylock函数 | 测试加锁，如果不成功则立即返回，错误码为EBUSY |

使用互斥锁前必须先进行初始化操作。初始化有两种方式，一种是静态赋值法，将宏结构常量PTHREAD\_MUTEX\_INITIALIZER赋给互斥锁，操作语句如下：

pthread\_mutex\_t mutex=PTHREAD\_MUTEX\_INITIALIZER;

另外一种方式是通过pthread\_mutex\_init函数初始化互斥锁，该函数原型如下：

int pthread\_mutex\_init (pthread\_mutex\_t \*mutex,const pthread\_mutexattr\_t

\*mutexattr);

函数中的参数mutexattr表示互斥锁的属性，如果为NULL则使用默认属性。互斥锁的属性及意义见下表：

|  |  |
| --- | --- |
| 属性值 | 意义 |
| PTHREAD\_MUTEX\_TIMED\_NP | 普通锁：当一个线程加锁后，其余请求锁的线程形成等待队列，解锁后按优先级获得锁 |
| PTHREAD\_MUTEX\_RECUSIVE\_NP | 嵌套锁：允许一个线程对同一个锁多次加锁，并通过多次unlock解锁。如果是不同线程请求，  则在解锁时重新竞争 |
| PTHREAD\_MUTEX\_ERRORCHECK\_NP | 检错锁：在同一个线程请求同一个锁的情况下，返回EDEADLK ， 否则执行的动作与类型  PTHREAD\_MUTEX\_TIMED\_NP |
| PTHREAD\_MUTEX\_ADAPTIVE\_NP | 适应锁：解锁后重新竞争 |

初始化以后，就可以给互斥锁加锁了。加锁有两个函数：pthread\_mutex\_lock()和pthread\_mutex\_trylock()。它们的原型如下：

int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex); int pthread\_mutex\_trylock(pthread\_mutex\_t \*mutex);

用pthread\_mutex\_lock()加锁时，如果mutex已经被锁住，当前尝试加锁的线

程就会阻塞，直到互斥锁被其他线程释放。当pthread\_mutex\_lock函数返回时，说明互斥锁已经被当前线程成功加锁。pthread\_mutex\_trylock函数则不同，如果 mutex已经被加锁，它将立即返回，返回的错误码为EBUSY，而不是阻塞等待。

注意：加锁时，不论哪种类型的锁，都不可能被两个不同的线程同时得到，其中一个必须等待解锁。在同一进程中的线程，如果加锁后没有解锁，则其他线程将无法再获得该锁。

函数pthread\_mutex\_unlock函数解锁时，要满足两个条件：一是互斥锁必须处于加锁状态，二是调用本函数的线程必须是给互斥加锁的线程。解锁后如果有其它线程在等待互斥锁，等待队列中的第一个线程将获得互斥锁。

当一个互斥锁使用完毕后，必须进行清除。清除互斥锁使用函数

pthread\_mutex\_destroy,该函数原型如下：

int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex);

清除一个互斥锁意味着释放它所占用的资源。清除锁时要求当前处于开放状态，若锁处于锁定状态，函数返回EBUSY，该函数成功执行时返回0。由于在Linux中，互斥锁并不占用内存，因此pthread\_mutex\_destroy（）除了解除到互斥锁的状态以外没有其他操作。

## 条件变量

条件变量是利用线程间共享的全局变量进行同步的一种机制。条件变量宏观上类似if语句，符合条件就能执行某段程序，否则只能等待条件成立。

使用条件变量主要包括两个动作：一个等待使用资源的线程等待“条件变量被设置为真”；另一个线程在使用完资源后“设置条件为真”，这样就可以保证线程间的同步了。这样就存在一个关键问题，就是要保证条件变量能被正确的修改，条件变量要受到特殊的保护，实际使用中互斥锁扮演着这样一个保护者的角色。Linux也提供了一系列对条件变量操作的函数，如下表所示：

操作条件变量的函数

|  |  |
| --- | --- |
| **函数** | **功能** |
| pthread\_cond\_init函数 | 初始化条件变量 |
| pthread\_cond\_wait函数 | 基于条件变量阻塞，无条件等待 |
| pthread\_cond\_timedwait函数 | 阻塞直到指定时间发生，计时等待 |
| pthread\_cond\_signal函数 | 解除特定线程的阻塞，存在多个等待线程时按入队顺序  激活其中一个 |
| pthread\_con\_broadcast函数 | 解除所有的线程阻塞 |
| pthread\_cond\_destroy函数 | 清除条件变量 |

与互斥锁一样，条件变量的初始化也有两种方式，一种是静态赋值法，将宏结构常量PTHREAD\_COND\_INITIALIZER赋予互斥锁，操作语句如下：

pthread\_cond\_t cond=PTHREAD\_COND\_INITIALIZER;

另一种方式是使用pthread\_cond\_init，它的原型如下：

int pthread\_cond\_init(pthread\_cond\_t \*cond,pthread\_condattr\_t \*cond\_attr);

其中，cond\_attr参数是条件变量的属性，由于其并没有得到实现，所以它的值通常是NULL。

等待条件成立有两个函数：pthread\_cond\_wait和pthread\_cond\_timedwait。它们的原型如下：

int pthread\_cond\_wait(pthread\_cond\_t \*cond,pthread\_mutex\_t \*mutex); int pthread\_cond\_timedwait（pthread\_cond\_t \*cond, pthread\_mutex\_t

\*mutex,const struct timespec \*abstime）;

pthread\_cond\_wait函数释放由mutex指向互斥锁，同时使当前线程关于cond指向的条件变量阻塞，直到条件被信号唤醒。通常条件表达式在互斥锁的保护下求值，如果条件表达式为假，那么线程基于条件变量阻塞。当一个线程改变条件变量的值时，条件变量获得一个信号，使得等待条件变量的线程退出阻塞状态。

pthread\_con\_timewait 函数和pthread\_cond\_wait 函数用法类似， 差别在于 pthread\_con\_timewait函数将阻塞直到条件变量获得信号或者经过abstime指定时间，也就是说，如果在给定时刻前条件没有满足，则返回ETIMEOUT，结束等待。

线 程 被 条 件 变 量 阻 塞 后 ， 可 通 过 函 数 pthread\_cond\_signal 和

pthread\_cond\_broadcast激活，它们的原型如下： int pthread\_cond\_signal(pthread\_con\_t \*cond);

int pthread\_cond\_broadcast(pthread\_cond\_t \*cond);

pthread\_cond\_signal()激活一个等待条件成立的进程，存在多少个等待线程时，按如对顺序激活其中一个；而pthread\_cond\_broadcast()则激活所有等待线程。

当一个条件变量不再使用时，需要将其清除。清除一个条件变量通过调用

pthread\_cond\_destroy()实现，函数原型如下：

int pthread\_cond\_destroy(pthread\_cond\_t \*cond)

pthread\_cond\_destroy函数清除由cond指向的条件变量。注意：只有在没有线程等待该条件变量的时候才能清除这个条件变量，否则返回EBUSY。

通过例5-3来演示条件变量的使用方法，在例子中，有两个线程被启动，并等待同一个条件变量。

例5-3 condition.c #include <stdio.h> #include <unistd.h> #include <pthread.h>

pthread\_mutex\_t mutex; pthread\_cond\_t cond;

void \*thread1(void \*arg)

{

pthread\_cleanup\_push (pthread\_mutex\_unlock, &mutex);

while(1) {

printf ("thread1 is running\n"); pthread\_mutex\_lock (&mutex); pthread\_cond\_wait (&cond, &mutex); printf ("thread1 applied the condition\n"); pthread\_mutex\_unlock (&mutex);

sleep (4);

}

pthread\_cleanup\_pop (0);

}

void \*thread2(void \*arg)

{

while(1) {

printf ("thread2 is running\n"); pthread\_mutex\_lock (&mutex); pthread\_cond\_wait (&cond, &mutex); printf ("thread2 applied the condition\n"); pthread\_mutex\_unlock (&mutex);

sleep (1);

}

}

int main(void)

{

pthread\_t tid1, tid2;

printf ("condition variable study! \n"); pthread\_mutex\_init (&mutex, NULL); pthread\_cond\_init (&cond, NULL);

pthread\_create (&tid1, NULL, (void \*) thread1, NULL); pthread\_create (&tid2, NULL, (void \*) thread2, NULL);

do {

pthread\_cond\_signal (&cond);

} while (1);

sleep (50);

pthread\_exit (0);

}

## 编译执行程序、结果是什么？？请分析thread1和thread2同步运行的情况。

## 

## 受到同一个变量的限制所以不能同时运行只能交替运行

1. **异步信号**

在Linux操作系统中，线程是在内核外实现的，它不像进程那样在内核中实现。Linux线程本质上是轻量级的进程。信号可以被进程用来进行互相通信，一个进程通过信号通知另一个进程发生了某事件，比如该进程所需要的输入数据已经就绪。线程同进程一样也可以接受和处理信号，信号也是一种线程同步手段。

信号（SIGINT和SIGIO）与任何线程都是异步的，也就是说信号到达线程的时间是不定的。如果有多个线程可以接受异步信号，则只有一个被选中，如果并发的多个同样的信号被送到一个进程，每一个将被不同的线程处理。如果所有的线程都屏蔽该信号，则这些信号将被挂起，直到有信号解除屏蔽来处理它们。

Linux多线程扩展函数中有三个函数用于处理异步信号： int pthread\_kill(pthread\_t threadid,int signo);

int pthread\_sigmask(int how,const sigset\_t \*newmask,sigset\_t \*oldmask); int sigwait(const sigset\_t \*set,int \*sig);

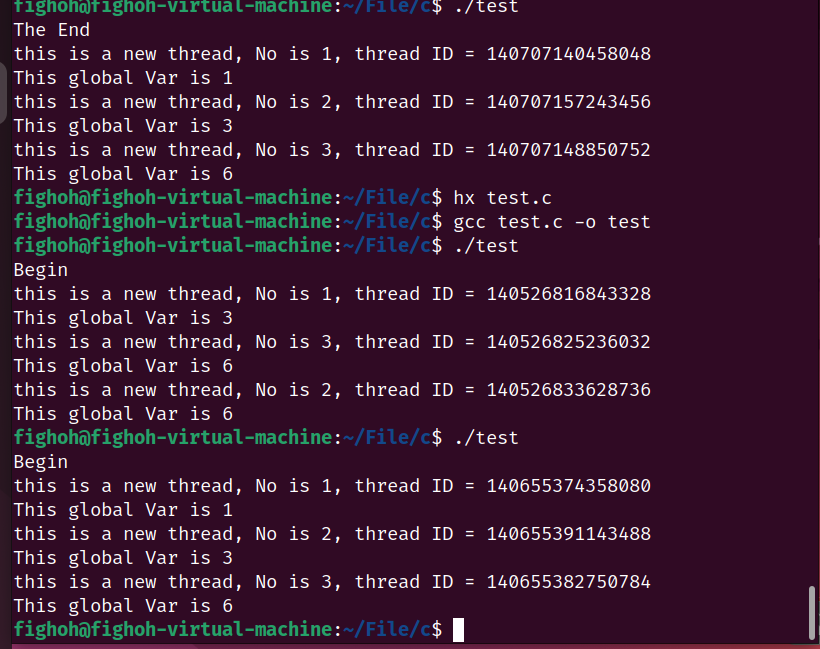
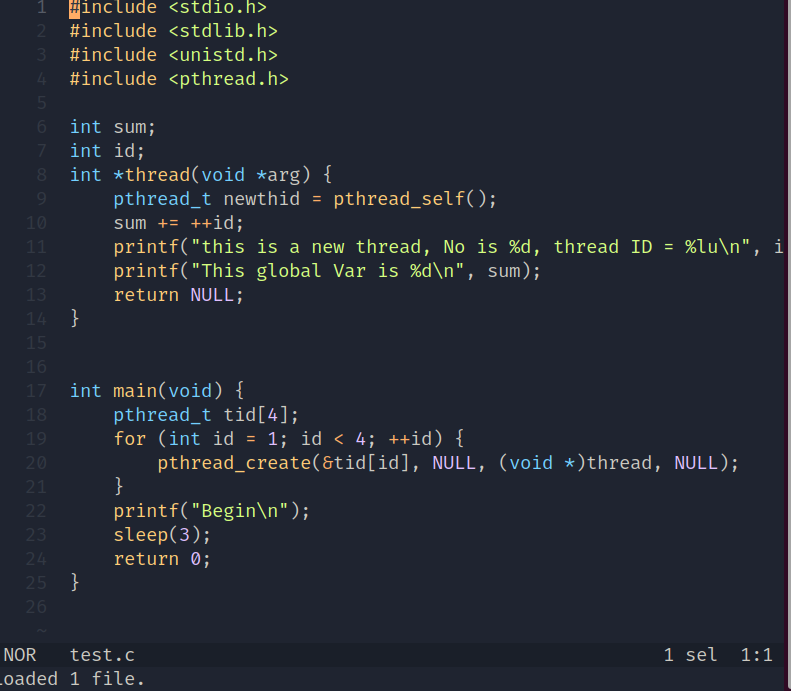
其中，函数 pthread\_kill 用 来 向 特 定 的 线 程 发 送 信 号 signo 。函数 pthread\_sigmask用来设置线程的信号屏蔽码，但对不允许屏蔽的Cancel信号和不允许响应的Restart信号进行了保护。函数sigwait用来阻塞线程，等待set中指定的信号之一到达，并将到达的信号存入\*sig中。

**作业**

1. **编写一个多线程程序：要求主线程创建3个子线程，3个子线程在执行时都修改一个它们的共享变量，观察共享变量的值，看看可以得出什么结论。**
2. **编写一个多进程多线程的程序：要求创建4个子进程，每个子进程都分别**

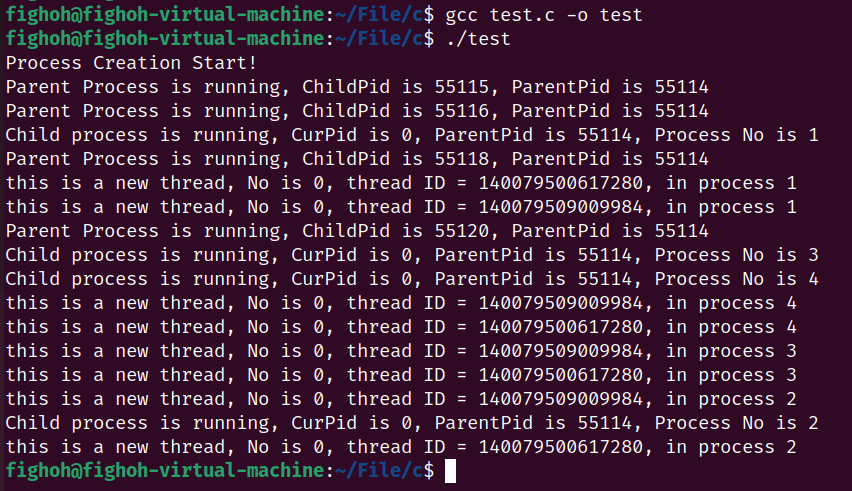
**创建两个线程，进程和线程的功能不做要求，可以只提供简单的打印语句。**

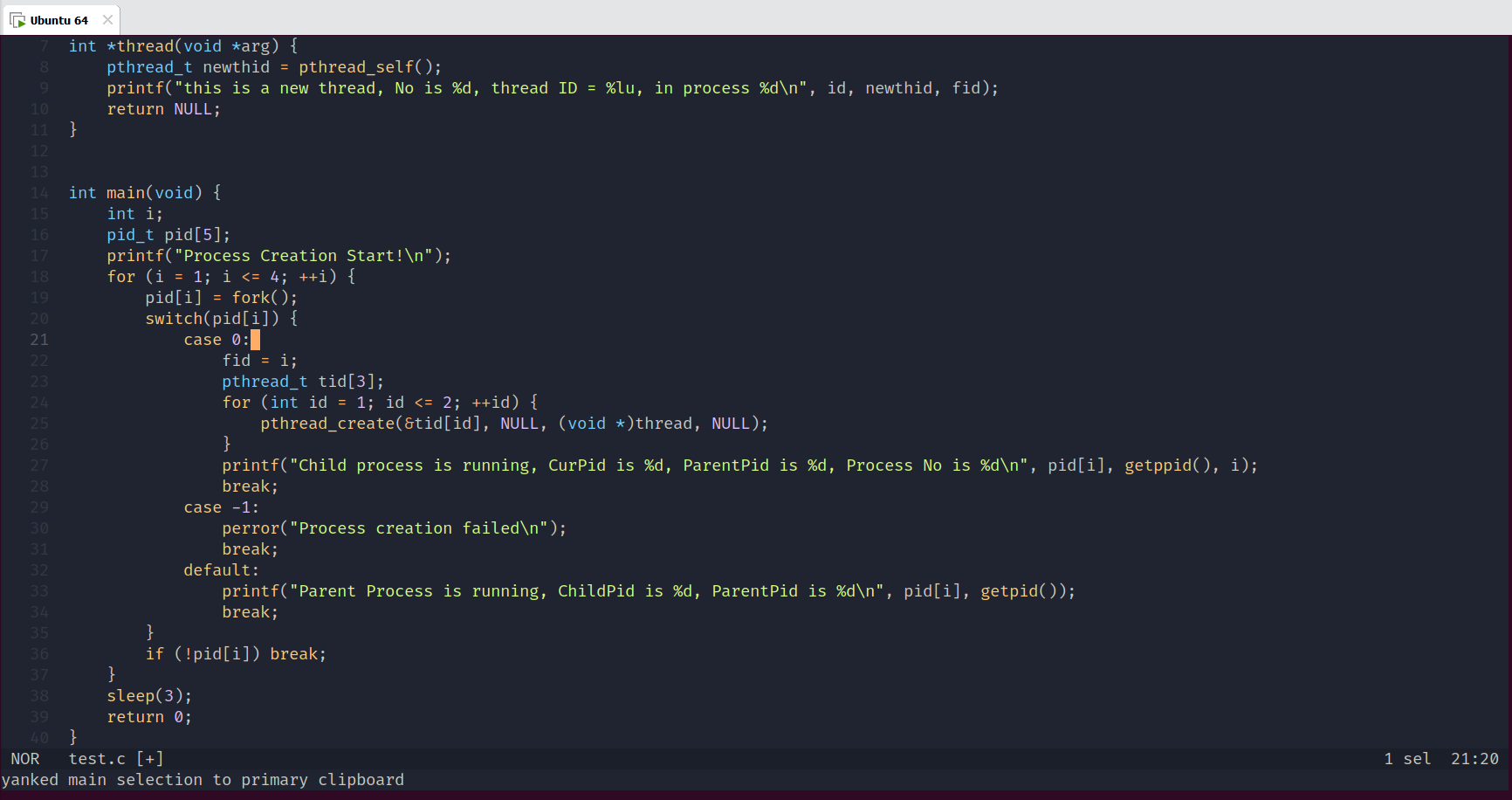
问题1：

****

线程可以共同修改这个变量并且是并发的。上图中可以看出那个global Var可能直接变成6。

问题二：





注意要用break打断循环并且利用共享的变量来控制生成的进程的数量，然后再在4个子进程内部生成两个线程，可以看出都是并发的